AFRL-IF-RS-TR-1999-265
Final Technical Report
January 2000

EMBEDDED GENETIC ALLOCATOR

BBN Technologies

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. F277

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

AFRL-IF-RS-TR-1999-265 has been reviewed and is approved for publication.

APPROVED: [illegible]

RALPH KOHLER
Project Engineer

FOR THE DIRECTOR: [illegible]

NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

EMBEDDED GENETIC ALLOCATOR

David B. Cousins
Fred Roeber

Contractor: BBN Technologies
Contract Number: F30602-97-C-0296
Effective Date of Contract: 9 September 1997
Contract Expiration Date: 8 September 1999
Short Title of Work: Embedded Genetic Allocator
Period of Work Covered: Sep 97 – Sep 99

Principal Investigator: David B. Cousins and Fred Roeber
Phone: (401) 848-3520 and 848-3548
AFRL Project Engineer: Ralph Kohler
Phone: (315) 330-2016

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Ralph Kohler, AFRL/IFTC, 26 Electronic Pky, Rome, NY.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE January 2000	3. REPORT TYPE AND DATES COVERED Final Sep 97 - Sep 99

4. TITLE AND SUBTITLE EMBEDDED GENETIC ALLOCATOR	5. FUNDING NUMBERS C - F30602-97-C-0296 PE - 62301E PR - D002 TA - 02 WU - P5
6. AUTHOR(S) David B. Cousins and Fred Roeber	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) BBN Technologies 10 Moulton Street Cambridge MA 02238	8. PERFORMING ORGANIZATION REPORT NUMBER NPT-122
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington VA 22203-1714 AFRL/IFTC 26 Electronic Pky Rome NY 13441-4514	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-1999-265
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Ralph Kohler/IFTC/(315) 330-2016	
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release, Distribution Unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This final report documents the work accomplished under the Embedded Genetic Allocator Program, a two year effort funded by DARPA/ITO. The major accomplishment of the program was to develop a new approach to the problem of automatically optimizing the use of memory and processor resources in high performance computing systems consisting of heterogeneous processor nodes connected on a high-speed interconnection fabric. This is frequently known as the mapping problem. The Embedded Genetic Allocation technology developed under this program can provide an automated mapping tool for the design and re-hosting of these systems.

The approach developed is automatic and broadly applicable to a wide variety of system architectures. It consists of a hybrid genetic algorithm optimizer (the Embedded Genetic Allocator or EGA), coupled with a software performance monitoring system (various ones can be used).

The results presented in this report demonstrate that the EGA can be used to optimize the allocation mappings real-world software, and that the resulting optimizations can rival or even improve upon those generated manually by a skilled programmer.

14. SUBJECT TERMS Automated Mapping Tool, Real Time Space-Time Adaptive Processing, NUMA System, Optimization of Memory Resources in Multiple DSP System, Optimization of Sonar Signal Processor			15. NUMBER OF PAGES 52 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

TABLE OF CONTENTS

9. FIGURES

Executive Overview

Abstract

This report describes the work accomplished under the Embedded Genetic Allocator Program, a two-year effort funded by DARPA ITO. The major accomplishment of the program was to develop a new approach to the problem of automatically optimizing the use of memory and processor resources in high performance computing systems consisting of heterogeneous processor nodes connected on a high-speed interconnection fabric. This is frequently known as the mapping problem. As systems and algorithms become more complex, the amount of engineering time required to optimally map software on an embedded system for performance, weight, and production cost will continue to increase. The Embedded Genetic Allocation technology developed under this program can provide an automated mapping tool for the design and re-hosting of these systems.

The approach developed is automatic and broadly applicable to a wide variety of system architectures. It consists of a hybrid genetic algorithm optimizer (the Embedded Genetic Allocator or EGA), coupled with a software performance monitoring system (various ones can be used). The EGA requires no programmer knowledge of the underlying non-uniform memory access (NUMA) architecture of the target hardware nodes or interconnection fabric, nor does it need to know the performance characteristics of the processor nodes (i.e. memory size, computation speed etc.). For memory allocation optimization, the programmer simply specifies the data buffers to be allocated and requires that certain groups of buffers share the same quality of performance. To perform processor allocation, the programmer writes the program using augmented MPI communicator calls and does not need to know the underlying multiprocessor interconnection details.

The EGA minimizes the execution time of time-critical portions of the target system in two ways: by generating allocations of target program data buffers to various banks of memory in the NUMA architecture, and generating allocations of processors to MPI communicators in an effort to minimize the execution time of time-critical portions of the target system. These trial allocations are loaded and evaluated directly on the target hardware. Measurements of process execution time are derived from accurate, synchronized event logging of multiple processors. The timing data provides the information for an "optimizer cost function" which the genetic algorithm uses when selecting from amongst competing allocations. Thus, the EGA automates the trial and error method of hand optimization.

The results presented in this report demonstrate that the EGA can be used to optimize the allocation mappings of real-world software, and that the resulting optimizations can rival or even improve upon those generated manually by a skilled programmer.

Chronology of Embedded Genetic Allocation Successes:

1998 Optimization of Memory Resources in Multiple DSP system

System Title: Digital Frequency Up-translator

Functional Description: Takes as input 5 digital signals at 2.5 kHz complex baseband. Combines the five bands into one 15 kHz band, and modulates the result to 120 kHz.

System Hardware Implementation: VME based Mizar DSP card. Three TI C40 DSPs used to up-translate 2 channels. Processors are interconnected with their COM ports. Total system has 36 channels for a total of 56 DSPs.

EGA optimized static placement of 192 data buffers throughout the Non-uniform Memory Access Architecture (9 banks of varying speed memory) using a smart-malloc call. Optimization goal was to minimize the latency of the first data buffer through the system pipeline.

Resulting Performance Improvement:

Human optimized version of code (hand placement of memory banks) took approximately 40 hrs, and resulted in a Latency time (the sum of execution time of three stages) of 1,886 uSec. Adapting the system for GA optimization took approximately 4 hours to set up. After 2.5 hours, the GA beat the human solution. It converged to its best solution in 17 hours, giving a score of 1,681 uSec (12% improvement). Thus better results were achieved with a factor of 10 less human effort, i.e. 10 × less effort for better results.

These results were published in the Conference Proceedings of the 1998 IEEE International Conference on Systems, Man and Cybernetics. Oct 11-14 1998

1998 Optimization of Sonar Signal Processor

System Title: Sonar Signal Processor

Functional Description: Takes as inputs 24x24 digital array channels. Performs matched filtering, beam-forming (spatial filtering) and detection processing on 32x32 output beams.

System Hardware Implementation: VME based Mercury RACEWAY. Thirty Intel i860 processors interconnected with a 160 MB/sec tree-switch interconnect.

System was built using MPI as the interprocessor parallelization paradigm. EGA optimized the processor allocation to each of three MPI communicators (one each for matched filtering, spatial filtering, and detection processing) by means of a library overloading the original MPI calls. Optimization goal was to minimize the total execution time of the system over 16x512 time samples.

Resulting Performance Improvement:

Human optimized version of code (hand allocation of processors to each of the three stages) was performed based upon an operation count for each of the three stages. Thirty processors were split evenly between the three stages 10/10/10, for an end to end execution time of 7.88 sec.

System was designed using MPI, which enabled the user to vary the allocation of tasks to processors. GA optimization took 1 hour to set up. The GA beat the human solution very quickly. It converged to its best solution after 6 hours, giving a score of 5.64 sec. (28% improvement). The final allocation was 11/15/5 with the processor allocation adjusted to take into account non-uniformity in the RACEWAY interconnection.

These results were published in the Proceedings of the High Performance Computing Symposium, 1999 Advanced Simulation Technologies Conference, The Society for Computer Simulation International, April 11-15 1999.

1999 Optimization of MITRE Real Time Space-Time Adaptive Processing Radar Benchmark.

System Title: RT_STAP Benchmark

Functional Description: System performs adaptive beamforming on Radar signals. Three stages of processing consist of channel equalization, doppler processing, and adaptive weights computation (QR decomposition).

System Hardware Implementation: Various VME based Mercury RACEWAY system. The systems were both homogeneous and heterogeneous and ranged from 26 to 128 node. Nodes used were based upon both i860 and PowerPC processors, interconnected with a 160 MB/sec tree-switch interconnect (both in RaceWay and MultiPort configurations).

System was built using MPI as the interprocessor parallelization paradigm. Initially the software did not support allocation of processors to three separate communicators, but instead used only one, so portions of the code were rewritten (about 300 lines of code out of a total of 6000). Future benchmarks using this communication library will not require code rewrites to be used with the EGA. The EGA optimized the processor allocation to each of three MPI communicators. Optimization goal was to minimize the total execution time of the system over 16 iterations.

Resulting Performance Improvement:

EGA showed repeatedly that it could meet or exceed the performance of a human allocation. Furthermore, it could map the RT_STAP program across heterogeneous nodes, taking into account differing processor performance for different functions, node memory sizes, and contention in the interprocessor communication switch network.

These results were presented at the 1999 High Performance Embedded Computing Workshop, Lincoln Labs, September 1999.

Future Work

The results herein show that EGA is capable of solving the mapping problem for systems of direct interest to the DoD. The resulting processor utilization figures have been shown to meet or outperform those generated by experts. To get even better utilization, future work should move to a function-placement / data-flow system. Though this type of system is even more difficult to code in the conventional manner, using the EGA as an optimizer for the internal mapping layer of a high level design environment is a very attractive alternative.

For example, consider using EGA to optimize the internal mapping layer of ISI's RT_ and HRT_express to auto-partition a real time system, and to optimize the placement of code on heterogeneous processors (such as adaptive computing elements or other hardware accelerators.

Finally, the idea of using EGA to control run-time reconfigurability for fault tolerance of remote systems is very attractive. As EGA deals with the system as a "black box", it does not need to know any details of the software or hardware failures except whether it performs its task correctly. Thus it can automatically generate mapping solutions that avoid bad hardware (such as memory, processors, data-path etc.) in the search for a new configuration which tolerates the degraded hardware.

Embedded Genetic Allocator for DSP Memory Allocation

Introduction

Allocating application processing to the various processors and memory banks in complex embedded multiprocessor systems to achieve optimum performance is an extremely difficult problem. This paper describes the first implementation of the Embedded Genetic Allocator (EGA); a system designed to solve this problem using a *genetic algorithm* (GA). The system evolves a design that meets application performance objectives by generating trial solutions to the problem and measuring the quality of each solution. This quality is determined through the use of real-time software monitoring. This monitoring uses a hybrid software- hardware approach to accurately measure application execution times across multiple processors while maintaining low intrusiveness in the software execution [1][2]. Because the allocation process is highly automated and based upon accurate performance data derived from actual system operation, optimal solutions to the allocation problem are reached much more quickly and efficiently than can be done using current, largely manual, approaches.

This stage of the EGA development focused on performing optimizations due to allocation of user data buffers to various kinds of memory banks in a NUMA system

The Problem Definition: Memory Allocation in a Multi-DSP System

Optimum Data Buffer Allocation in a Real-Time Multi-Processor NUMA System

When developing a real-time application on a multi-processor NUMA system, there are a number of factors that can affect application execution speed including processor types, memory types, concurrent utilization of memory and bus resources, and hierarchical interconnection schemes. These factors all affect application execution in ways that are difficult to model and hence predict. Even when these effects can be measured, it can be difficult for a programmer to use this information to minimize the combined effect of all the factors on execution time.

As a concrete example, a description of the target architecture used in the experiment follows. The target hardware is a VME based quad Texas Instruments C40 DSP board manufactured by Mizar Inc. (Model MZ7772) [3]. The memory hierarchy on this board consists of four distinct banks of memory, each with different performance levels ranging from small and fast on chip SRAM to large and slow globally shared DRAM (see Figure 1 for details).

The choices a programmer makes in selecting where data and intermediate results are stored can have a large impact on program performance, often by an order of magnitude or more. Most DSPs use the Harvard bus architecture, which has multiple buses, allowing the CPU to retrieve data from two different memory banks in parallel. Furthermore, instructions are used that can fetch two pieces of data and perform a mathematical

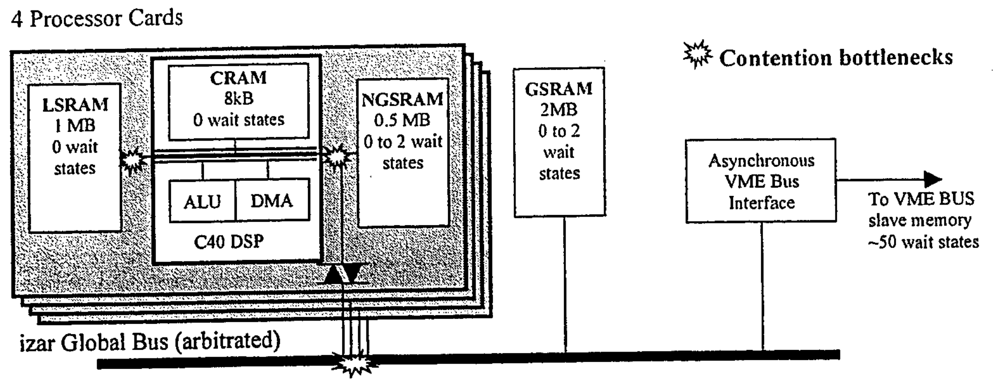

4 Processor Cards
LSRAM 1 MB 0 wait states
CRAM 8kB 0 wait states
ALU
DMA
C40 DSP
NGSRAM 0.5 MB 0 to 2 wait states
GSRAM 2MB 0 to 2 wait states
Contention bottlenecks
Asynchronous VME Bus Interface
To VME BUS slave memory ~50 wait states
izar Global Bus (arbitrated)

Figure 1: Target NUMA Architecture (Mizar MZ7772 VME board)

calculation all in one clock cycle. Figure 2 indicates an optimal memory mapping of a typical DSP algorithm (a convolution or FIR filter) which allows maximal use of the various DSP data paths.

A more naive implementation which utilizes only one bank of memory for both the program code and data, will exhibit poor performance due to excessive contention for the single memory bank. The situation can be exacerbated by the use of on-chip DMA engines which are commonly used to move input and output data between processors and I/O subsystems, causing further contention on the various data buses available to the system. Performance can be improved by separating data that is retrieved in parallel into different banks. Another obvious improvement is to place the buffers that require the greatest amount of access into the fastest memory banks, although these may not be large enough to allow this in some memory-restricted architectures. Thus, the buffer allocation problem can be viewed as a combination of an optimal packing problem and a contention reduction problem.

The manual approach to optimizing the data buffer allocation is commonly done by trying various placements of buffers to different memory banks and measuring the resulting execution time of the software. Many iterations are usually necessary to optimize performance. The difficulty of the problem rapidly increases with the number of

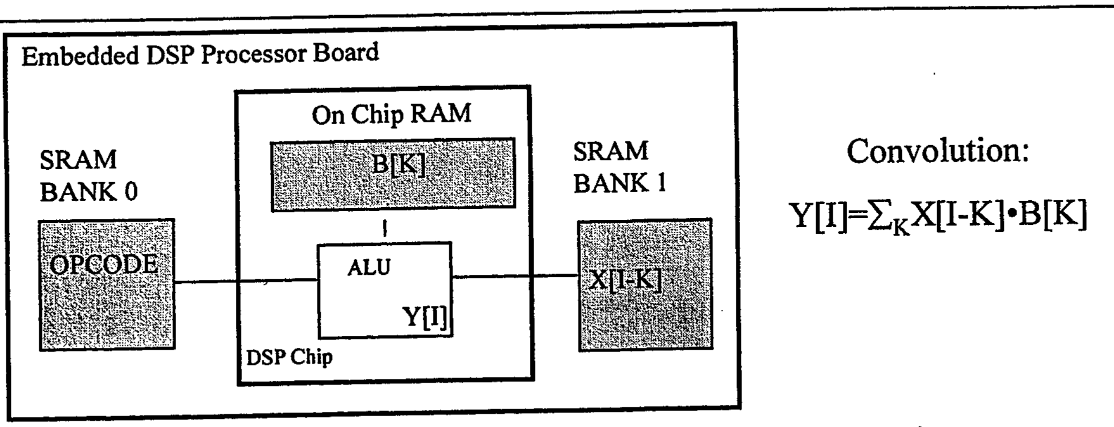


Figure 2 : Optimal Memory Allocation for a Typical DSP Function

buffers that need to be allocated. With many modern applications requiring hundreds of buffers, the manual approach is beyond what can be effectively solved by a human, requiring the use of sub-optimal techniques such as local optimization of key execution loops, rather than the entire program.

However, several general automated search methods can be effectively employed to find optimal resource allocations. The EGA system uses a genetic algorithm to search for an optimal buffer allocation.

The Embedded Genetic Allocator

A Description of the EGA System

The EGA system manipulates a population of trial "system design blueprints"; each representing an allocation of the software's data buffers among the various memory banks in the system. These trial blueprints are translated by EGA into input files that are automatically loaded by the target system under optimization. Run-time event collection generates performance data that the EGA uses to score the trial allocation. A genetic algorithm (GA) is used to generate new trial blueprints based on the feedback provided by the performance evaluation.

The EGA system consists of several software components (see Figure 3). Two of these components are embedded in the software being optimized. The first is an Online Memory Allocation Library (OMALib) which the programmer uses to identify the key data in the program. The second is an event logging library (ELOG) that is used to instrument key portions of the code for run-time performance monitoring. The remaining components of the EGA system run on a host workstation connected to the target system. The first host component is a Genetic Algorithm Engine which generates the trial allocations. The second is a Target Evaluation System that executes the trials on the target architecture and mediates communication between the target system, the GA Engine, and the user via a graphical interface.

Target-Based EGA Components

In the EGA system, the programmer manually identifies the key data components of the target code through the use of calls to the Online Memory Allocation Library (OMALib). This library is linked into the target code and acts like a smart "malloc". It allows the user to identify the data buffers to be allocated and specify any restrictions of

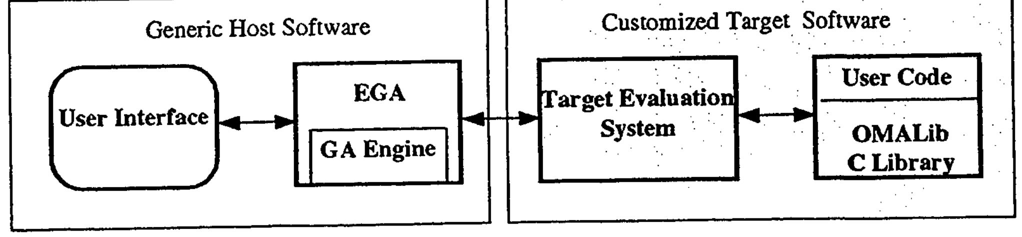


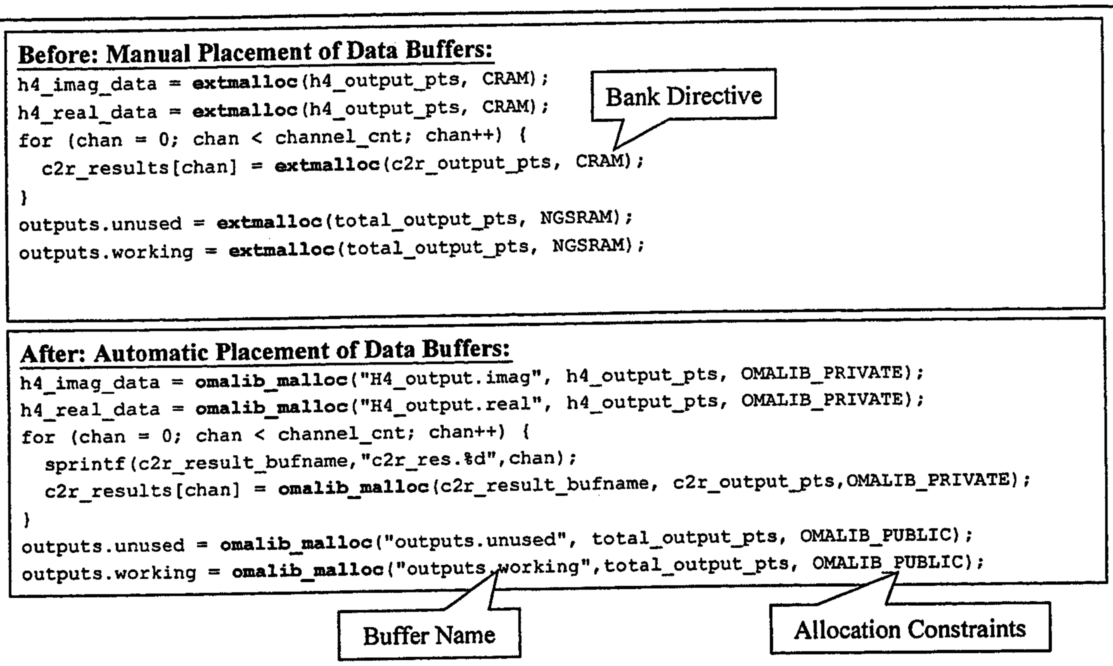
Figure 3:EGA Software Architecture

the allocation (i.e. the buffer must be allocated in shared memory or aligned on an address boundary). A section of sample DSP code is shown in Figure 3 both before and after use of OMALib.

OMALib runs in two modes. The *configuration* mode is used to generate a profile of the data buffer allocation required by the target system. In this mode, each call to the OMALib malloc generates data that is used to define the system. In *run* mode, each malloc call looks up the appropriate allocation for each buffer from a table provided by the GA Engine.

The OMALib allows the user to define two types of groupings of the data buffers (see Figure 4). These groupings provide two benefits: they are used by the EGA to reduce the dimensionality of the optimization search, and they provide equal quality of performance for certain related data buffers. The first type is called a *Process Group*. This defines a collection of software processes in the system that runs the same executable code. All processes in a Process Group share the same data buffer allocation scheme. The second type of grouping is a *Buffer Group*. When buffers are allocated via OMALib's malloc call, they must be uniquely named. The naming convention supports a group name identifying the buffer as being a member of a Buffer Group. All the buffers in a group are then allocated as a single larger buffer by the EGA. For example, all the buffers in a ping-pong data buffer would be grouped together, as would all the history buffers in a bank of filters operating on multiple channels. Both processor and buffer groupings significantly reduce the number of discrete buffers to be allocated.

In addition to using the OMALib, the user must also instrument the target code with an Event logging library (ELOG). The user brackets key portions of their code with function calls that generate time-stamped *events* that are used to derive performance

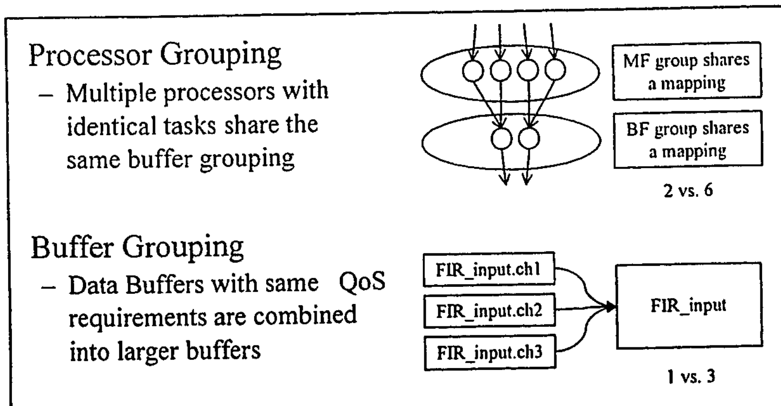

Before: Manual Placement of Data Buffers:

```
h4_imag_data = extmalloc(h4_output_pts, CRAM);
h4_real_data = extmalloc(h4_output_pts, CRAM);
for (chan = 0; chan < channel_cnt; chan++) {
  c2r_results[chan] = extmalloc(c2r_output_pts, CRAM);
}
outputs.unused = extmalloc(total_output_pts, NGSRAM);
outputs.working = extmalloc(total_output_pts, NGSRAM);
```

After: Automatic Placement of Data Buffers:

```
h4_imag_data = omalib_malloc("H4_output.imag", h4_output_pts, OMALIB_PRIVATE);
h4_real_data = omalib_malloc("H4_output.real", h4_output_pts, OMALIB_PRIVATE);
for (chan = 0; chan < channel_cnt; chan++) {
  sprintf(c2r_result_bufname,"c2r_res.%d",chan);
  c2r_results[chan] = omalib_malloc(c2r_result_bufname, c2r_output_pts,OMALIB_PRIVATE);
}
outputs.unused = omalib_malloc("outputs.unused", total_output_pts, OMALIB_PUBLIC);
outputs.working = omalib_malloc("outputs.working",total_output_pts, OMALIB_PUBLIC);
```

Figure 4 : Sample DSP memory allocation code before and after use of OMALib



Processor Grouping
- Multiple processors with identical tasks share the same buffer grouping

Buffer Grouping
- Data Buffers with same QoS requirements are combined into larger buffers

Figure 5: User defines groups in the code by using process and buffer names

statistics about the software. The overhead of logging an event is minimal (<10 usec) due to the use of an external monitoring system known as TraceMaker™ which performs synchronized time-stamp management and off-loads all the event logging overhead from the target processors. The TraceMaker device gathers this event data and generates an event log file on the host system. Details about how EGA utilizes this log file to minimize the execution time of the target system are given in a later section.

Host-Based EGA Components

There are two host-based components of the EGA: the Target Evaluation System (TES) and the Genetic Algorithm (GA) Engine. The TES is an Expectk program that controls the target system processes and mediates communication between those processes and the GA Engine. The engine is a C++ program that creates trial allocation blueprints using GA techniques (described later in more detail). A Graphic User Interface (GUI), written in Tcl/Tk, enables the user to configure and run the system and monitor the status of an optimization run.

When the GA Engine creates a trial allocation, it performs a simple design rule check to eliminate allocations that are invalid due to memory bank size restrictions. Allocations that fail this check receive a very poor score and are not tested on the target system. If the trial passes, the TES runs the target code on the target system. The OMALib library uses this memory allocation scheme for the run, and the ELOG library sends performance data to the TraceMaker system. The TES monitors both the target code and the TraceMaker system and notifies the GA Engine upon run completion. The GA Engine then uses the TraceMaker event log file to generate a *fitness* score (also known as the optimizer cost function) for that allocation. The whole process iterates as the GA Engine generates new trials, until one of the convergence criteria is met. The three criteria that may be specified by the user are: 1) a target fitness score is achieved, 2) a fixed number of iterations has elapsed, or 3) the population has converged (the fitness of the best allocation found so far has not changed by a specified percentage within a specified number of iterations).

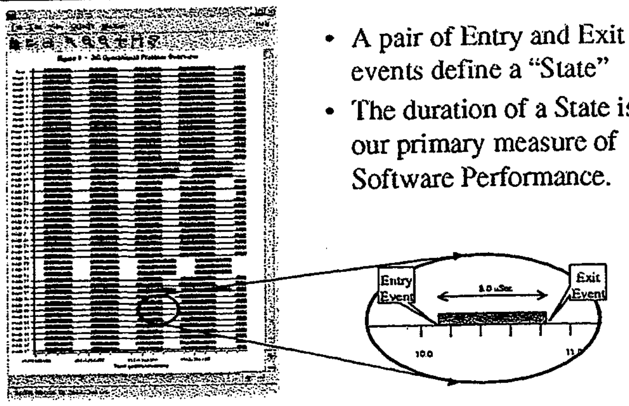
Generating the Fitness Score (Cost Function)

In general, an optimization requires a single measure to be maximized or minimized. This value is referred to interchangeably as both a cost function or a fitness score in the remainder of this paper. In the case of the EGA, optimizing for maximum software performance is equivalent to minimizing the software's execution time as measured by the real-time event monitoring system. This section describes how these measurements are used to derive a single fitness score for a trial allocation.

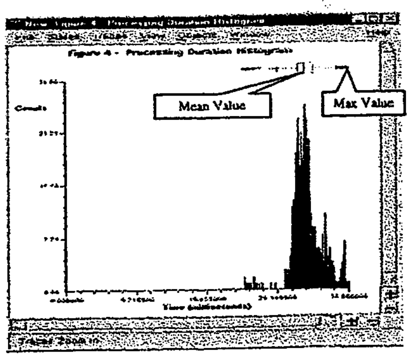
Each process in the user's target system is instrumented to log events that are collected and time-stamped by the BBN TraceMaker. All the events produced by a given DSP are stored together in a *trace*. Furthermore, the user defines *states* by denoting pairs of starting and stopping events for each state (see Figure 5 for a sample trace). States are used to measure the execution time of loops or function calls in the target code under optimization. The user's program should iterate over these functions enough times to generate good statistics for the state durations. The TraceMaker device collects the traces generated by each processor and generates an *event log* that is written to the host computer's disk.

The GA Engine uses the event log to calculate the durations of the states in all the traces. It then calculates the overall fitness score of the run by combining the state timing information according to user specified criteria that determine which statistical and combinatorial operators are used for the calculations.

The timings for all the instances of a state within a single trace are combined into a single state value by taking either the average, sum, minimum, maximum, or standard deviation of all the state durations in that trace. The resulting values for each of the states in a trace are then combined using one of the aforementioned operations to produce a single value for each trace. Finally, all the trace values are combined into a single fitness score in a similar way. Thus, state durations may be used to optimize for minimum execution time, minimum data transfer latency, etc. In our experiment, we compute the average execution time for each of the three DSP processing loops that compose the system, and then minimize the sum of the three values. In this particular system, the execution times of the three processing loops were independent of each other, so the



- A pair of Entry and Exit events define a "State"
- The duration of a State is our primary measure of Software Performance.



- A Histogram of a State's duration collected over a run provides the optimization "objective function"
  - Mean duration
  - Minimum
  - Maximum
  - Variance etc.
- State statistics are combined in various ways to measure characteristics such as latency and throughput

Figure 6 : Multiprocessor Event Logs Provide the Raw Performance Data From Which EGA Derives Performance Statistics

minimum of the sum should be equivalent to the sum of the minimums.

A Brief Description of Genetic Algorithm-Based Optimization

The GA approach to optimization is based on automatically generating and evaluating different solutions to the problem at hand in a controlled fashion until a solution satisfying the desired constraints is found. The GA was initially inspired by Darwinian evolution and is in a sense an emulation of biological evolution. There exist several summary articles and texts regarding GAs although there appears to be much controversy among researchers as to the relative importance of the various techniques [4][5][6][7]. A brief description of the GA techniques used in the EGA follows.

The EGA produces and evaluates a trial memory allocation during each *iteration* of its GA (see Figure 7) for details. Once an initial population of randomly generated allocation schemes is created, the GA produces new allocations schemes by either combining portions of two previously generated schemes (the *crossover* operation) or by randomly perturbing an existing scheme (the *mutation* operation). If the new memory allocation has a better evaluation than the worst scheme in the population, then the new scheme replaces the worst in the population.

The EGA implements *Darwinian Selection* where schemes in the population with better evaluations have a greater chance of being selected for crossover and mutation operations. It is this selective pressure that helps drive the convergence of the population. The GA ranks the *N* schemes in the population from most fit *(n=0)* to least fit *(n=N-1)*. The probability, P_n, of a scheme being selected as a parent of the next trial depends on its ranking in the population, *n*, and is given by

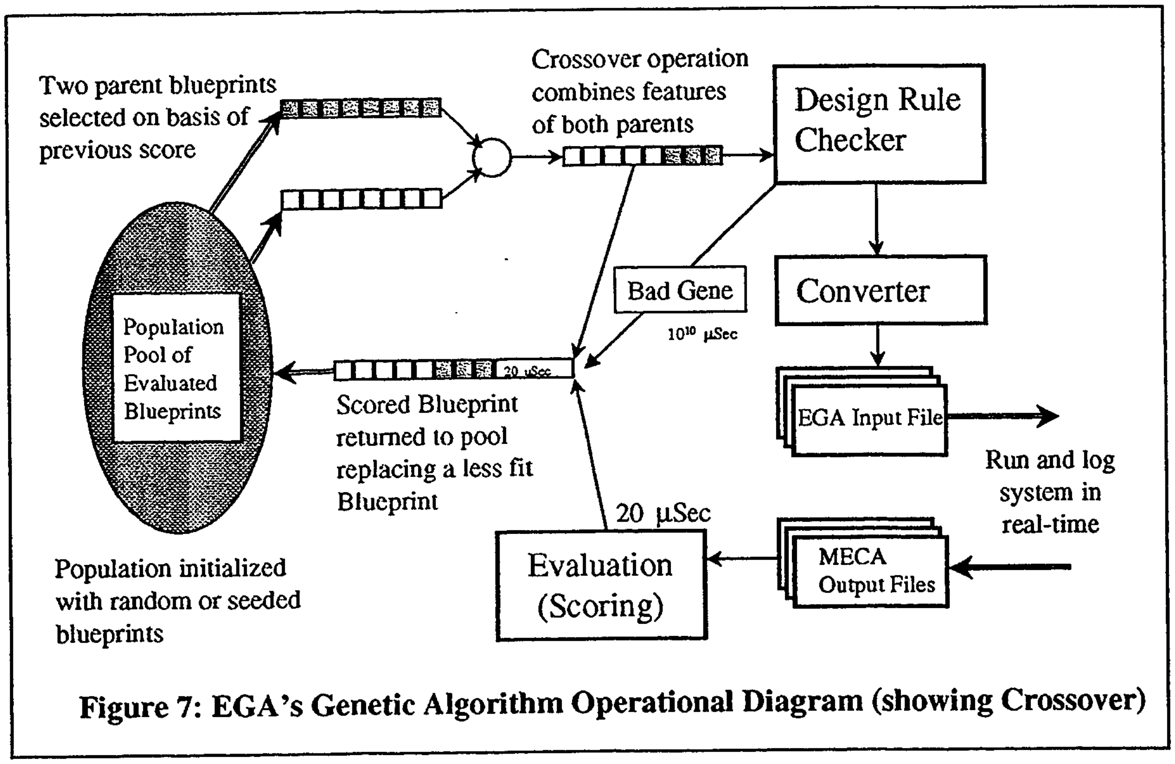

Two parent blueprints selected on basis of previous score
Crossover operation combines features of both parents
Design Rule Checker
Bad Gene
10^{10} µSec
Converter
Population Pool of Evaluated Blueprints
20 µSec
Scored Blueprint returned to pool replacing a less fit Blueprint
EGA Input File
Run and log system in real-time
20 µSec
Population initialized with random or seeded blueprints
Evaluation (Scoring)
MECA Output Files

Figure 7: EGA's Genetic Algorithm Operational Diagram (showing Crossover)

$$P_n = x^n \frac{1-x}{1-x^N}$$

where x is called the *parent scalar* value ($0 < x < 1$). Each scheme is x times as likely to be a parent as the scheme ranked next higher in the population. The value of x is chosen so that there is some chance for the lower fitness schemes to be selected before the end of the optimization session; otherwise the population would become quickly dominated by the best individuals. The EGA uses the heuristic for setting the parent scalar: $x = a^{1/p}$ where p is the population size, and a is the ratio of P_{N-1} to P_0. In our experiments, $a = 1/20$ has worked very well for the population sizes with which we experimented (100 to 800 members).

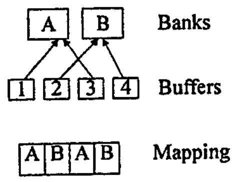
EGA Blueprint Encoding Methods

The trial solutions to the buffer allocation problem need to be encoded in a form that the GA can manipulate. This encoding is commonly referred to as a *chromosome*, which gets manipulated by the crossover and mutation operations. The EGA has been implemented using two different methods of encoding chromosomes. The first method is *integer encoding,* where each element in an integer array corresponds to one of the buffers the EGA allocates (see Figure 8). The integer value of the element specifies the memory bank into which the buffer is allocated. This direct encoding offers good performance and a simple implementation.

The second encoding scheme, *order encoding*, more closely incorporates the heuristics used by the programmer during hand optimization (see Figure 9). In this method, the buffers are numbered from 1 to N where N is the number of buffers to be allocated. Additionally, the integers $N+1$ to $N+M-1$, where M is the number of memory banks, are used as tokens to control the allocation algorithm. These numbers are placed in an arbitrary order, creating an ordered list.

A modified *greedy* algorithm is used to convert the ordered list of buffer numbers into an allocation using the algorithm shown in Figure 9. The bank tokens are required in order to allow the greedy algorithm to encode all possible allocation schemes. To see this, consider a simple case with only two buffers, both of which suffer contention with each

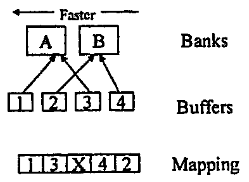
Good performance
Simple Implementation



```
or I = 1..N {
Bank_Index = gene(I)
```

Figure 8: Integer Encoding of Chromosomes

- Uses Ordered Chromosome operators
- Modified Greedy Algorithm
- Much better performance
- Complex Implementation



```
For I = 1..N+1 {
  if gene(I)=='X' {
        disable further
        allocation to bank A
  }else{
        allocate gene(I) to best
        bank that can hold it
  }
}
```

Uses knowledge of bank "quality" to generate "better" solutions

Figure 9: Order Encoding of Chromosomes

other. Furthermore, let the fastest memory bank be large enough to store both buffers. The optimal allocation places these buffers in two different banks. If the algorithm did not have the ability to turn off allocation to the fastest bank, it would never find this optimal solution. Note that the algorithm will never turn off the last memory bank. The last bank is usually the slowest/largest available. The bank is guaranteed to be large enough to hold all the buffers because OMALib places all the buffers in this bank when the target is first run in configuration mode.

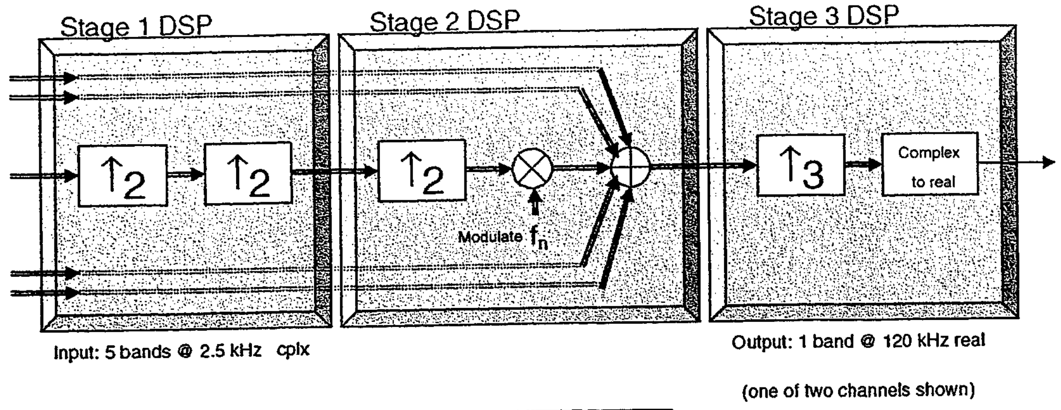
The EGA does not allow duplicate chromosomes in the population in order to help maintain the population diversity and to prevent a single good chromosome from dominating the population. In the case of ordered chromosomes many chromosomes will encode the same allocation scheme since the order in which the buffers get placed in a bank does not effect program performance. To improve convergence performance, the EGA requires that the population contain unique allocation schemes, eliminating multiple entries that result in the same allocation. This offers a significant reduction in the problem space since for N buffers allocated into M banks there are N^M unique allocations (hence integer chromosomes) and $(N+M-1)!$ unique ordered chromosomes. For large N we have $(N+M-1)! >> N^M$.

Experimental results (see below) have indicated that the ordered encoding method yields a faster convergence rate, giving well scoring allocations faster than the integer encoding method. This is intuitively satisfying because there is greater chance of fully utilizing the fastest banks.

Demonstration of Memory Allocation Optimization – Experimental Results

Description of Experimental Target System

As a proof of concept, the EGA was used to optimize buffer placement in an existing real-time multiprocessor system. The target system is a subset (two channels out of 36) of a larger system designed to up-sample base-band acoustic data to the ultrasonic operating frequency of an unmanned underwater vehicle. The original system was implemented using seven processor boards identical to those depicted in Figure 1. The experimental target used one of these boards, with three processors each performing a stage of the signal processing chain and a fourth generating synthetic data and checking the resulting output for correctness. The on-chip DMA engines are used to pass data between the processors via serial ports connecting the processors.


Stage 1 DSP
Stage 2 DSP
Stage 3 DSP
↑2
↑2
↑2
Modulate f_n
↑3
Complex to real
Input: 5 bands @ 2.5 kHz cplx
Output: 1 band @ 120 kHz real
(one of two channels shown)

EGA performs optimized memory allocation:

stage 1	110 buffers in 17 groups
stage 2	57 buffers in 8 groups
stage3	25 buffers in 12 groups
total:	192 buffers in 37 groups

Grouping reduces chromosome size from 192 to 37

Figure 10 :Demonstration Test Case

The software for the test system was taken directly from the existing system, modified slightly to allow stand-alone operation, and instrumented with the OMALib and ELOG libraries. The signal processing performed by the three stages is shown in Figure 10 along with a table of the number of buffers to be allocated for each processor. Notice that buffer grouping reduces the number of buffers to be allocated from 192 to 37, thereby reducing the search space for the global optimization (from $4^{192} \approx 10^{115}$ to $4^{37} \approx 10^{22}$ possible allocation schemes).

The original code was hand optimized by the designer using TraceMaker event logging to monitor timing and synchronization. The approach used by the expert to tune the performance of the system was to define and optimize buffer allocations internally in the source code of each DSP library function (such as the interpolation filter function or modulator) and thereby use these same allocations for all instantiations of the functions in the system. This *local* optimization of functions is a standard technique of managing complex software problems. Because the EGA is able manage a much larger number of variables than the human designer, it performs *global* optimization and evaluates complex allocation schemes that the human does not have the resources to develop.

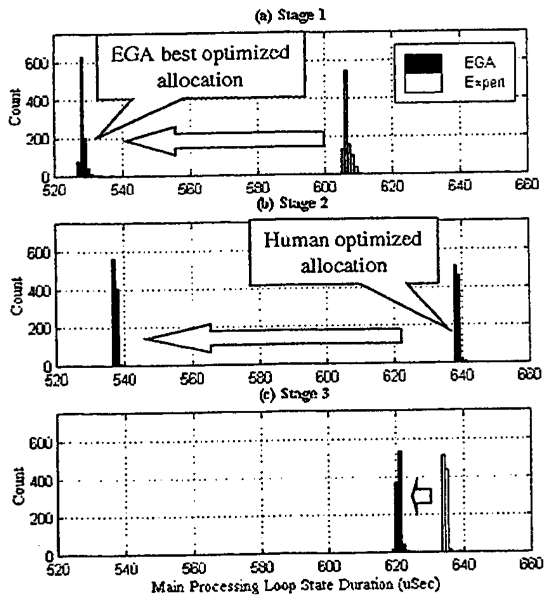
Comparison of EGA results with human expert solution.

The EGA optimization runs used a population size of 600 with the relative probability of performing the mutation operator set to 40% and crossover 60%. The fitness score of each trial allocation was the sum of the average execution times for each of the three component stages as mentioned previously. The runs were set to terminate after 5000 iterations, approximately 24 hours of computation time. The best EGA buffer allocation resulted in a 1,681 μsec fitness score, exceeding the 1,886 μsec performance achieved by the expert human allocation by 12%. The worst allocation tried by the EGA gave a score of 14,248 µsec, showing the order of magnitude variation in performance due to data buffer location in the DSP NUMA system.

Figures4a through 4c show state-duration histograms for the main execution loops for each of the three stages of the target system. The original expert human allocation is shown as white histograms, and the best EGA solution as dark histograms. In all cases, the EGA solution shows better performance (although the third stage shows only small improvement). Notice that in all cases the histograms are tightly clustered; the rare occasional outliers are due to the event logging over the asynchronous VME bus. The large shift in the means of the histograms between the two sets of allocations indicates that the improvement is statistically significant. In all fairness, the overall throughput improvement of the system is limited by the stage with the longest execution time, so the overall throughput was only improved by 15 µsec. However, at this point, the designer can reconsider the partitioning of the work between the stages in order to bring the three stages into a more balanced configuration. Future research in the EGA will address automating this procedure as well.

EGA Convergence Performance

Figure 11 shows the performance of an EGA population for a pair of typical runs with the settings described in the previous section. The dashed lines indicate the best and worst members of a 600 member population for a run using integer encoding. The solid lines show the same for an ordered encoding run. As noted previously, the ordered encoding reaches better solutions faster than the integer encoding due to the inclusion of



Human optimization time:	~ 40 hours
Human setup time of EGA:	~ 4 hours
EGA beats human score	~ 2.5 hours
EGA Converges	~ 16 hours

==> 10 × less effort for better results.

<u>Initial Latency (sum of three stages)</u>:
Human Score = 1,886 uSec
EGA Score = 1,681 uSec
EGA is 12 % better

<u>Throughput (longest stage):</u>
Human Score = 635 uSec
EGA Score = 620 uSec
EGA is 2 % better (need to repartition)

Figure 11: Histograms Showing Speedup Due to EGA Allocation

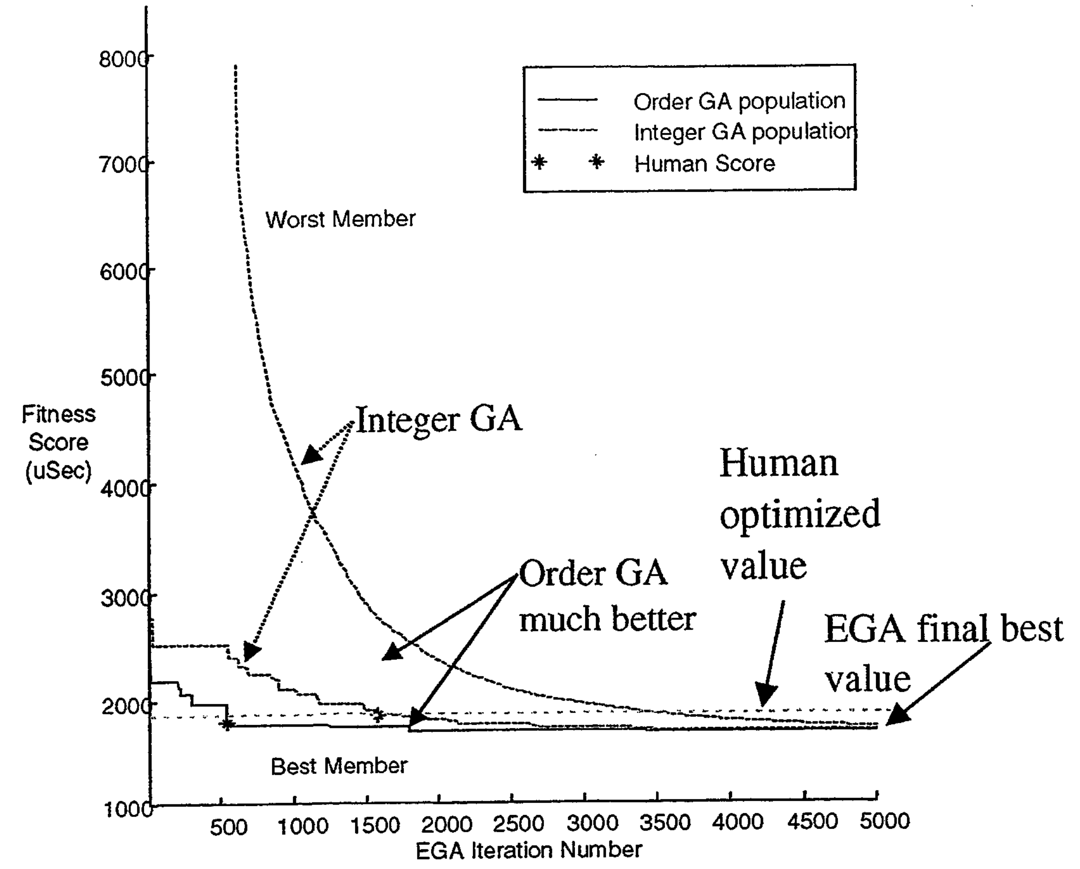


Figure 12 : EGA Convergence Performance

more domain knowledge in the algorithm.

The point at which each run surpasses the human optimized system performance is also indicated. Thus even though the entire run took 24 hours of clock time to perform, good solutions appear after only 500 iterations of the ordered algorithm (after about 2.5 hours).

It should be noted that initially the population is quite poor because it is generated randomly. However, it quickly converges until all members of the population are better than the human allocation. This large number of near-optimal allocations indicates that this target problem is resource-rich in memory and that many unique allocations give similar good results.

Practically speaking, we are concerned only with the best member of the population, and how fast we reach the globally optimal solution. The chosen size of the population directly effects the convergence rate. A good value for the EGA population size is largely determined by the difficulty of the problem being optimized. In a sense, the EGA performs many searches in parallel with each search centered on a member of the population. The more difficult the problem, the larger the population size needs to be in order to have a high probability of finding the global minimum. The population size of

600 for the experimental problem was chosen by trial and error, monitoring the convergence rate of the EGA and adjusting the population size accordingly. The population size was considered sufficient if after repeating a run ten times, the converged population found the global minimum in every run.

Lessons Learned

The lessons learned in this stage of the project can be summarized as follows:

Careful engineering of the GA chromosome encoding yields best performance payoff. This was demonstrated by the superior performance of the ordered gene feeding the "greedy with denial" allocation algorithm, as compared to the brute-force integer chromosome.

Grouping of buffers by processor and data quality-of-service significantly reduces solution time. This form of simplification will indeed reduce the available number of solutions. However, in many circumstances, like QoS constraints "make sense" in parallel code where the following cases occur: multiple processors at a barrier synchronization having to wait for the longest process, and pipelines where the overall throughput is governed by the slowest stage.

Acceptable tuning of GA parameters can be achieved by using a "similar model" problem in software. Thus thousands of iterations per second can be generated, as opposed to 0.1 iteration per second. Tuning GA performance is still a black art, and is highly problem specific. Skillful selection of a "homologous problem" that shares the combinatorial complexity of the real problem will often provide good insight to the selection of GA parameters.

"Local" library optimizations by users (or library designers) result in sub-optimal performance. "Global" optimization by EGA can uncover new system performance constraints that can then be manually re-partitioned by user. Often times, the reasons for the performance improvement of the new optimization is difficult for even an expert to understand.

EGA for Multiprocessor Allocation for MPI based Systems

Introduction

This next section of the report addresses the allocation of application processing to the various processors in complex multiprocessor systems to achieve optimum performance is an extremely difficult problem. This section describes the second implementation of the Embedded Genetic Allocator (EGA) which was developed to address this problem.

Development of the EGA during this phase of the program focused on performing optimizations in the following two categories: 1) optimizing the usage of inter-processor communications bandwidth on large distributed architectures by adjusting the mapping of software processes to hardware processors, and 2) application specific parallelization parameters that the designer uses to divide and map the problem onto the processor pool in various ways.

The Problem Definition: Optimum Process Placement in a Real-Time, Heterogeneous, Multiprocessor System

When developing applications for multi-processing systems, the software design is driven both by algorithmic considerations and by system performance considerations. With modern multi-stage interconnect systems (e.g. Mercury's RACEWAY, Sky's SkyChannel, and the Intel Paragon) the effect of inter-processor data routing on system performance is often difficult, if not impossible, to predict in large systems.

As a concrete example, consider a representative Simple Signal Processing (SSP) application. Data from *M channels* are fed through a simple time domain matched filter using an overlap-add FFT technique (FFT length *N*). All channels of each sample time are then fed through a simple 2D FFT beam former, producing *B beams*. The beam data are input to a simple threshold detector that sets thresholds in each beam by examining adjacent beams. The exact details of the processing are not pertinent to the discussion. (See Figure 1 for details).

The mapping of this algorithm onto P processors involves the consideration of several tradeoffs, such as the size of the individual tasks, the relative speed of the processors, the interconnect, and the location of processors on the interconnect topology. Task size is chosen to maximize processor usage and minimize idle time. The ability to keep these processors busy is driven by the ability of the interconnect to move input and output data from functional units operating on different processors. The balance between I/O and task size complements interconnect bandwidth with processor speed. Proper placement of functional components on the multiprocessor can reduce the amount of communication necessary by confining intermediate results to a single processor. Many modern multi-processor connection fabrics divide the processor pool into sub-groups in such a way that communication with processors that are close in an interconnect sense is preferable (in terms of contention and speed) to communication with more remote processors. This affects function placement by imposing a preferred assignment so that functions that must share data should be located "close" to one another.

Let us use the SSP as an example to illustrate the tradeoffs inherent in the process placement. Dividing the filter processing into M parallel filters appears to be simple – each task works on one input channel. Similarly, one can divide the beamformers into S tasks, each generating all the beams for one time sample. Finally, we break the detectors into B parallel tasks, again one per beam. Thus, we would have M + S + B = T parallel tasks.

Several problems arise with this assignment. First, the samples from each of the M filter tasks must be routed to all of the B beamformer tasks. Additionally, since many multiprocessor connection fabrics are most efficient transferring contiguous blocks of data (as opposed to single data words), for certain systems it will be far more efficient to have the beamformer tasks work on a contiguous block of samples (as opposed to single time slices). To simplify programming, this block size is often tied to the size of the FFT used for the matched filter convolution (usually *N* divided by a power of two), and is chosen to correspond to an efficient transfer block size. Therefore, the number of parallel tasks for beamforming becomes $S = N/2^i$, where i is a design parameter. Finally, the output of the beamformer tasks must each be sent to some of the B detectors, since these detectors use neighboring beams for threshold setting (in the most extreme case, all the beams would be used for each detection). Again, for performance and programming simplicity, this is best done in blocks. An engineer with some knowledge of the system can make reasonable estimates for these processing parameters. However, optimum selections of these values will vary from machine to machine, and usually have to be determined by manually timing the tasks and hand-tuning the values.

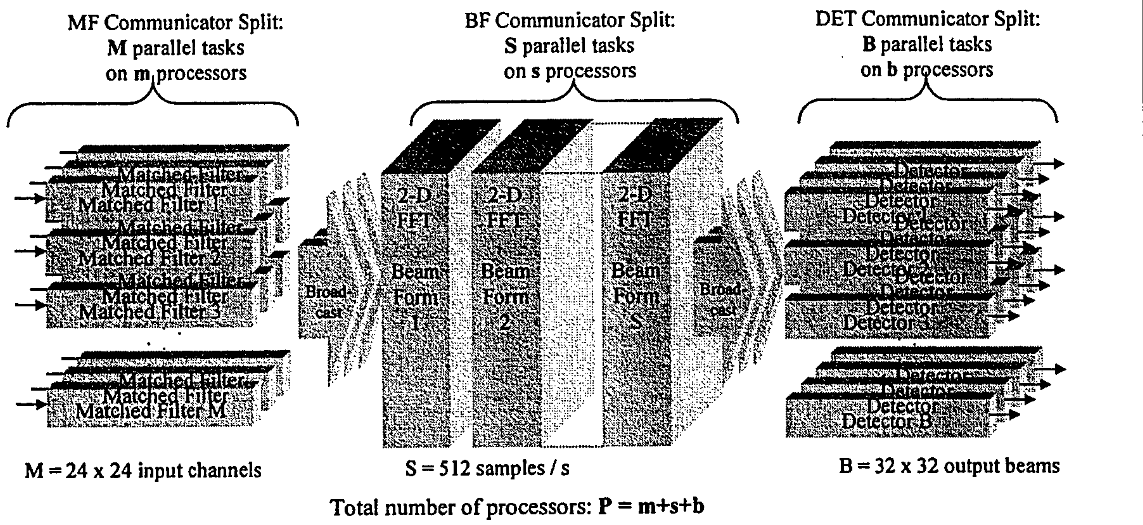


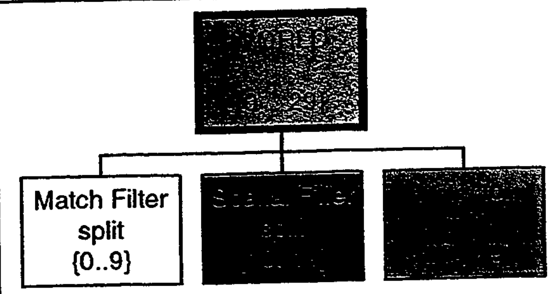
Figure 13 :Simple Signal Processor Application. A pipeline consisting of three groups (splits) of processors.

Several programming paradigms exist for implementing the parallelization. In order to maximize the portability of the solution, MPI (Message Passing Interface) was chosen as the underlying structure for segmenting the parallel code functionality. MPI is a standard library and is available in public domain implementations and as part of many high performance computing operating systems*.

A central notion to MPI is the idea of a *communicator*, which is a collection of processes that are capable of performing both point to point (i.e. send and receive) and collective operations (i.e. barrier synchronization, data broadcast, and data reduction). Each communicator *ranks* its processors from 0 to N. The numbering of a specific processor in one communicator need not have any relationship to that in another. MPI provides for creating a communicator whose processor membership is comprised of a sub-set or *sub* of another communicator. MPI also provides a routine for *splitting* a communicator into some number of mutually exclusive sub-sets of processors according to a key called *color.* The root of all communicators is a pre-defined communicator consisting of all processors entitled the *world* communicator.

For the SSP example case, a single call to the MPI_comm_split function is used to divide all the available processes into 3 sub-groups (MPI colors) of the MPI world communicator (see Figure 14). These are termed the Matched Filter (MF), Beam-forming (BF), and Detection (Det) groups. Each group is responsible for executing one stage of the processing. The number of processors in each group are indicated by m, s, and b in Figure 13. MPI takes care of all the task assignment overhead, allocating the M matched filter tasks to the group of m processors. Similarly s processors do S beamforming tasks, and b processors do B detector tasks (in Figure 14, m=s=b=10).

The EGA will be used to change the size of the individual groups of processors as well as their membership, automating performance optimization using both load balancing and reduction of interconnection bandwidth contention.



```
WORLD 30
Split WORLD SSP_TypeMFilt <freedom> {0 1 2 3 4 5 6 7 8 9}
Split WORLD SSP_TypeSFilt <freedom> {10 11 12 13 14 15 16 17 18 19
Split WORLD SSP_Type.Detect <freedom> {20 21 22 23 24 25 26 27 28
```

Figure 14: MPI Communicators for SSP (showing resulting EGA file parsing)

The EGA for Processor Allocation

Modifications from the EGA Memory Allocation System

The EGA for Processor Allocation has essentially the same basic structure as the EGA built for Memory Allocation. The initial design intended the two systems to be merged, but because a RISC processor was chosen as the target for our processor allocation experiments (as opposed to a NUMA DSP) this was deemed an

* See http://www.mcs.anl.gov/Projects/mpi/index.html for a list of references to the various implementations of MPI that are available.

unnecessary complication. The fundamental difference in the two systems is that one uses augmented MPI calls for processor allocation, and the other uses augmented malloc calls for memory allocation. Additionally, the chromosome representation for processor allocation is superficially similar to that of memory allocation. However, the details of the representation are quite different, and allow for representation of very complex systems. The remainder of this section focuses on the new system components, as well as the chromosome representation.

None	Reorder	Reselect	Resize
World	World	World	World
{0 1 2 3}	{0 1 2 3}	{0 1 2 3}	{0 1 2 3}
↓	↓	↓	↓
Sub	Sub	Sub	Sub
{0 2 3}	{3 0 2}	{2 0 1}	{2 1}

Figure 15: Examples of Sub Processor Allocation Under Different EGA Freedoms

Target-Based EGA Components

For portability, the EGA optimization capability is implemented as an augmentation on top of standard MPI. The user source code interface consists of wrappers around MPI communicator calls that assign processors to task groups according to an EGA-generated trial allocation (these trials are frequently referred to as *chromosomes* because they are generated by a Genetic Algorithm). Utility functions are provided for implementing an optimization loop containing the key code to be timed. Each loop iteration uses an input chromosome that directs a new processor assignment, resulting in different execution performance. The augmented communicator creation calls operate in two distinct modes. On the first iteration, the system is in an *initialization* mode. Here, calls to the augmented MPI communicator routines produce the same results as normal MPI calls would, generating a file that describes the initial default MPI processor allocations. This file also includes user-supplied flags that specify the amount of *freedom* allowed the EGA when it generates new trial allocations (see below). On subsequent iterations, the system works in a *runtime* mode, where the augmented MPI code redefines the communicator membership based on trial chromosomes generated by the GA Engine on the host system.

To constrain the optimization problem (thus speeding up the convergence to a near-optimal solution), the user can specify the degrees of freedom allowed to the EGA when manipulating the processor membership in a communicator. Three degrees of freedom are allowed (see Figure 15):

1) The lowest level of freedom is *re-order*. Re-ordering keeps processor membership constant within a communicator but permutes the processor's *rank*. This can optimize process placement within a multiprocessor and is most valuable when data transfer times within the interprocessor connection fabric are non-uniform.

2) The next freedom level is *re-select*. Re-selection keeps the communicator size constant, and varies processor membership. Re-selection is most useful when there are multiple communicators of fixed size whose processor membership may or may not overlap. It is also useful for redistributing the data transfer load within the connection fabric in order to avoid congestion. Re-selection implies re-ordering.

3) The most versatile level of freedom is *re-size*. Re-sizing allows communicator size to vary. Resizing allows the EGA to optimize both load balancing among the available processors and utilization of the interprocessor connection fabric topology.

As before, the user must use either an Event logging library or (if the execution time is large enough) a pair of operating system timer calls bracketing the key section of code to generate execution time data. This data is then used by EGA for feedback on trial allocation performance.

```
for each unique communicator W:{
  k = P+(C_W-1)
  g(1..k) = next substring of chromosome
  Select 1st color in communicator as current color
  for i = 1 to k{
    if (g(i) < P){
      Assign processor g(i) to current color
    } else {
      g(i) is a token indicating color change
      increment current color to next color
    }
  }
}
```

Figure 16: MPI Chromosome encoding

Host-Based EGA Components

The user front end to the Processor Allocation EGA is essentially unchanged from the one used in the Memory Allocation experiments. Internally, the TES was redesigned in order to make the system more portable, and to simplify using the EGA with pre-existing systems.

When the GA Engine creates a trial chromosome, it performs a simple design rule check to eliminate allocations that are invalid due to physical constraints (for example, one with no processors allocated to one of the communicators). Allocations failing this check receive a poor score and are not tested on the target system. If the trial chromosome passes the design rule then TES runs the target code on the target system. The MPI library uses this processor allocation scheme for the run, and performance data is generated either from calls to the ELOG library or from the operating system timer. From this point, interpretation of the event data and generation of the fitness score (cost function) is identical to that done for memory allocation.

EGA Blueprint Encoding Methods

An ordered chromosome is used to represent internally the processor allocation to the user's MPI communicators. This representation is a sequence of unique numbers. MPI sub-communicators are treated as MPI split-communicators with only one color.

The EGA reads in the file containing the description of the MPI structure defined in the user's software. All communicators with the same parent are grouped and sorted. Given P processors and W communicators with C_W colors in each communicator, there are W substrings in the chromosome, each with $P+C_w-1$ elements. Each position in the substring nominally corresponds to a position in the processor list of the corresponding communicator, with the value at that position corresponding to a unique processor ID number. However, the extra C_W-1 elements in the substring act as tokens used to delineate the division of processors among the colors of split communicators when resize is enabled. As a specific example, for the 30 processor SSP problem, P=30, W=1, C_1=3, so there are 32 elements in the chromosome representing the problem.

The algorithm used by the EGA to allocate processors based on this chromosome is summarized in the listing of Figure 16.

Demonstration of MPI Process Placement – Experimental Results

Description of Experimental Target System

As a proof of concept, the EGA was used to optimize a pipelined implementation of the SSP algorithm described earlier. In this case, there were M = 576 channels (a 24x24 sensor array), each complex base-banded upon input. There are B = 1024 beams (32x32 2DFFT) and the convolution frame size was 512 samples. The processing evaluation input data consisted of 16 such frames.

The target system was a 32 node Mercury Computer i860 VME RACEWAY system running MC/OS 4.3*, and consisting of 2 MCV9 boards each with 16 Intel i860 processors and an ILKVSB to connect the two boards together. Due to previous system failure, two of the processors in this system were not operable: the 6th and the 32nd. The host was a Motorola MVME162 VME single board computer with a 25MHz 68040 processor and 2 Mbytes of RAM. This host was running VxWorks 5.2. The compilation host for the Mercury processors was a Sun UltraSparc. The Hughes implementation of MPI V3.3 for RACEWAY was graciously supplied for this effort.

As mentioned previously, the EGA augmented MPI_comm_split function was used to divide the 30 available processes into three sub-groups (MF, BF, and Det). EGA was allowed to vary the size and membership for these sub-groups over all possible combinations. Some resulting configurations could not be tested on the hardware due to memory constraints on the processor nodes of the hardware (i.e. too many channels fill all available processor memory). These "invalid" configurations are trapped at initialization and assigned poor scores. For each chromosome, per-processor initialization was done, 16 iterations (512 sample frames) of the processing were executed, followed by per-processor cleanup. The score was derived from the total execution time of the 16 processing iterations: the initialization and cleanup time were not part of the score.

The "human" partition cited in the results was based upon a quick FLOP computation. The human programmer divided the processors into three equal groups, i.e. 10 processors allocated per group.

A large part of the software design and implementation effort involved setting up the communication between the sub-groups after the adaptive split had been determined. For instance, the output of each Matched Filter processor must be sent to all Beam-former

* MC/OS 4.3 was provided for this study by Mercury Computer Systems, Inc. 199 Riverneck Road, Chelmsford, MA 01824 USA

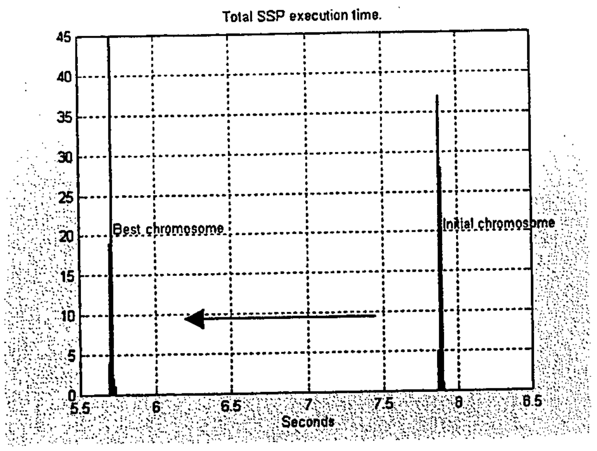


Figure 17: A Comparison of Histograms from Human and Best Allocations

processors. This was done using the MPI Broadcast routine, which requires the creation of a new communicator for each Matched Filter processor that includes that Matched Filter processor and all of the Beam-formers. An identical method was used between the Beam-formers and the Detectors. As an aside, we later found that the implementation of the standard MPI_Bcast routines that were used unfortunately do not take advantage of the RACEWAY broadcast capability. Thus, the communication transfers between the three stages execute slower than we expected.

Comparison of EGA results with human expert solution.

The EGA optimization runs used a population size of 200 with a 20% probability of mutation and 80% probability of crossover. The fitness score of each trial allocation was the average execution time for completing the entire algorithm. The runs were set to terminate after 5000 iterations, approximately 24 hours of real time. The initial human expert allocation had an execution time of 7.88 seconds ($m = s = b = 10$ processors in Figure 1). The execution time of the best EGA allocation was 5.64 seconds for a split with 11 MF, 15 BF, and 5 Det processors (i.e. $m = 11$, $s = 15$ and $b = 5$ in Figure 1). This represents a speed improvement of 28%. The worst execution time was 57.3 seconds, (allocating a single processor in the beamforming communicator, $s = 1$). This indicates that bad processor allocations can give an order of magnitude variation in performance.

The initial human allocation was made without regard to the underlying topology of the interconnection fabric and simply assigned contiguous process numbers into communicators (as is often done with MPI programs). The EGA solution distributed the allocations over the two cards in a manner that balanced communication load as well as processing load: Det processors were all allocated to one VME card, MF processors were evenly split between both VME cards, and BF processors were allocated to the remainder of the available processors.

Histograms displaying the statistical separation between the initial human and best EGA scores are shown in Figure 17. The data obtained from the event log files are binned into 0.01-second intervals. Each histogram represents about 200 iterations.

EGA Convergence Performance

Figure 18 shows the performance of a 200 member EGA population for a typical run with the settings described in the previous section. The plot shows the time evolution of the scores (execution times) for the best and worst members of the population, as compared with the score of the initial human expert allocation. The plot is truncated after the 3500th iteration, as the system does not improve after that point.

The population is seeded with 200 random allocations. It should be noted that initially, most of the population is quite poor. However, the best member of this random seeding still beats the human's allocation score. However, the total population improves rapidly, with all 200 members beating the human score by the 1000th iteration. It can be seen from the plot that the best score keeps improving incrementally until reaching its final value after about 3600 iterations.

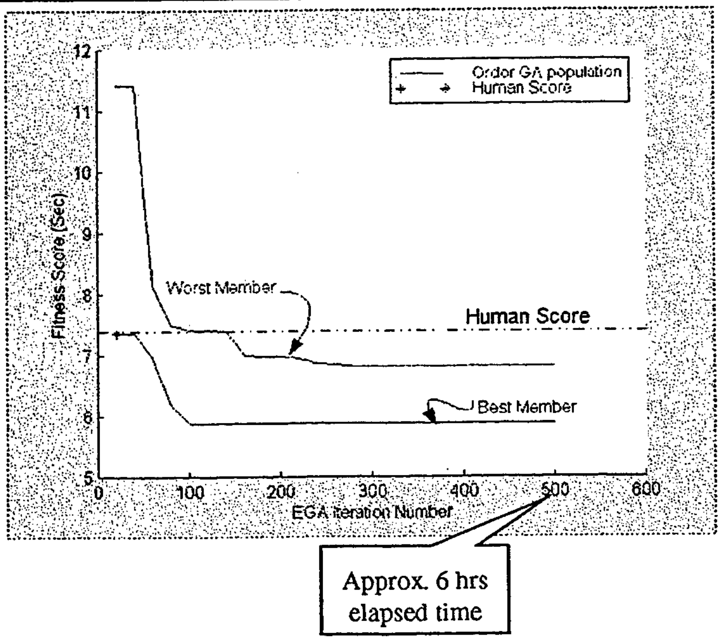
Lessons Learned

The lessons learned in this stage of the project can be summarized as follows:

Automatic process mapping and load balancing of an MPI based system can be achieved by EGA using an augmented MPI communicator call. However, most existing MPI code is not written to take this extra degree of flexibility into account, but rather rely upon the assumption of the MPI communicator's default process allocation.

Parallel Systems with pipe-lined structure are very conducive to EGA optimization via load balancing.

EGA finds the biggest speed-up by repartitioning the processors and then fine-tunes result by shuffling processor allocation within communicators. For homogeneous processor systems, this results in smoothing out the interconnection hotspots. Heterogeneous processor systems (as are examined in the next section) will result in much larger variance of performance due to this reshuffling, and as such result in



•Finds optimum after ~100 Iterations (~1 hr)

•EGA beats human by **27%**

•Best gene is 50% faster than worst in population (indicating a wide range of "naïve" allocations)

•Worst case (57 sec) had single processor doing Beamforming

Figure 18: EGA SSP Experiment - Convergence Properties

simultaneous resizing and shuffling.

Optimizing at the process level restricts possible optimization of processor utilization. To do better, one must move down to placement at the functional level. We believe that use of a data-flow architecture will allow this to be done.

EGA for Optimization of Parallel Systems using the DARPA Data-Reorg API on Heterogeneous Platforms.

Introduction

In a logical follow on to the work presented in the previous section, the EGA system was applied to a large scale problem currently of interest to our DARPA and AFRL sponsors. This problem is the real-time implementation of a Space-Time Adaptive Processor (STAP) for generating high resolution radar images. The implementation of the STAP algorithm that was chosen for optimization was the Real Time STAP Benchmark (RT_STAP) written by MITRE. This program is written using MPI as the underlying interprocessor communications layer, and as such was an ideal choice for EGA optimization. We shall see that in the course of this effort, we uncovered several important reasons why direct optimization of the program's underlying MPI communicators is not a good idea. The effort was redirected and the result was a system that extended the Data Reorganization API (a layer in RT_STAP built upon MPI communicators that deals with all the intricacies of mapping and moving portions of the 3D radar data cube between processors). These extensions allow a program using the Data Reorganization API to be mapped to the underlying hardware in a far more flexible manner than was previously possible.

Problem Definition: Modifying RT_STAP and the MITRE Data-Reorg API to support EGA Optimization

The RT_STAP benchmark provided EGA with the opportunity to optimize a known representative application with online self-validation. RT_STAP was independently developed by another group, so there could be no coding style biases. This was somewhat of a concern in the previous experiment where the SSP was written from scratch to support explicit processor mapping.

The benchmark uses MPI collective operations to implement a "Clique" oriented algorithm: all processors execute each stage in a sequential manner. We shall see that using this organization generated one of the main difficulties with using EGA to automatically optimize this code.

The RT_STAP algorithm is an example of a "grand challenge" problem for embedded super-computing. Figure 19 shows how the program is organized into three

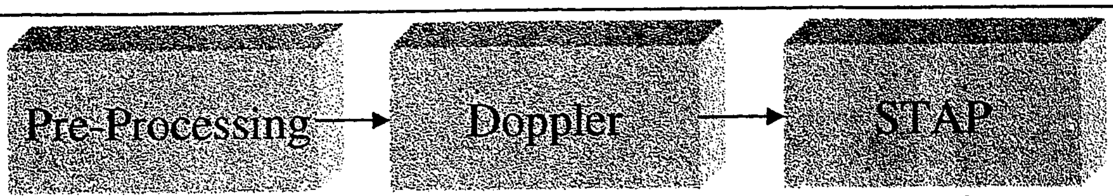


Figure 19 - Three stage structure of RT_STAP showing the three communicators

stages of processing. The first stage (pre-processor) is highly vector oriented and is implemented in a vector library. The second stage (doppler processing) is memory intensive as it gathers the data from the first stage corner turn. It also uses the vector library heavily. The last stage (QR-decomposition) is written in looped C code, as the vector library does not have the required functions. The output of one stage is the input to the next. The nature of the processing is such that data reorganization takes place at each of these transitions, termed a "cornerturn". This cornerturn is implemented using a generic data reorganization (Data Reorg) library, which supports data distribution and movement amongst a set of processors.

Figure 20 shows a TraceMaker event log of a Sample Clique mapping of RT_STAP onto a 26 node heterogeneous Mercury RaceWay Multicomputer consisting of 10 PPC nodes and 16 i860 nodes. The three stages are all color-coded. Dedicated data reorg calls (in this case built upon the MPI collective all-to-all operation to do corner turns) impose barrier synchronization points which restrict processor utilization and disallow overlap of data transfer with processing.

One can see the white areas where processors are waiting for the corner turns to finish. We can compute the processor utilization by measuring the amount of white space in the plot. The reasons this occurs are twofold.

1. The execution speed of each stage is limited to the most heavily loaded processor at that stage. If the number of processors does not evenly divide the number of smallest processing block, there is wasted processor time. The limit here is the benchmark restriction of 64 processors for the STAP stage. Additionally, in a heterogeneous system some processors do a given task better than others do. In Clique processing there is no provision made for matching processor type to job.

2. All of the inter-processor communication occurs simultaneously, and are synchronization points that limit processor utilization.

The resulting performance is 16 Iterations executed in 6.87 seconds with a processor utilization of approximately 53%

The initial efforts to optimize RT_STAP's under-lying MPI communicators directly did not meet with success. First, there were too many assumptions made in the program about the implicit rank ordering of default communicators, so re-arranging that ordering causes program bugs. Writing around this requires a significant rewrite to the entire program. Secondly, as mentioned before, the implicit structure of the RT_STAP program is a "Clique" oriented one. Basically this means that all processors must participate in performing each stage of the processing. This restricts the allowable EGA optimizations to simply reordering all the processors in each of the three communicators. While minor improvement may be gained, Clique processing is not conducive to heterogeneous load balancing due to the homogeneity of the implicit task allocation. Finally, as seen in Figure 20, the conventional practice of using MPI collective ops restricts the available processor loading, limiting the flexibility needed for EGA optimization.

However, a single solution was found that dealt with all of the above problems. The solution was to add the flexibility of process re-distribution to the Data Reorg library, where it is hidden from the user along with all the other complexities inherent in data redistribution. This modification is discussed in the next section.

Modifications to RT_STAP and the Data-Reorg Layer

To provide EGA with sufficient flexibility to optimize execution time, the data reorganization library supplied with the RT_STAP benchmark was modified to allow for the specification of different groups of processors for each stage of processing. These groups are specified on the command line as ranges of processor rank, and can be disjoint, overlapping, or identical. Using these command line arguments along with correct process creation scripts; any allocation can be realized. This also required modification to the main routine so that certain processing was requisite upon membership in the various stages. The additional specification of process distribution is done via MPI_Groups. Additionally, a slight modification of MITRE Data Reorganization Library allows the support of bipartite corner turns (where one group of processors performs an all-to-all with another group of processors).

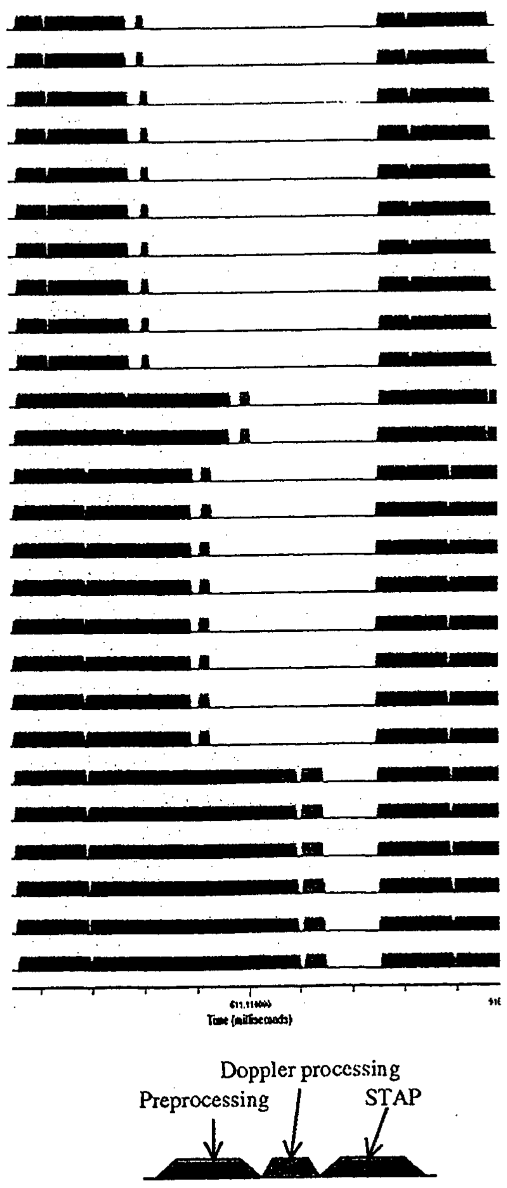


Figure 20-Event Log of Clique RT_STAP Showing Processor Under-utilization

All in all, the modifications accounted for about 7% of the total benchmark code, most of which is restricted to the data reorganization library and is thus re-usable. These modifications addressed some of the above shortcomings in the benchmark. Summarizing the key points of the result:

1. Each stage can now be parallelized individually. This allows for the application of up to 2880 processors to the problem, although the STAP stage is still limited to 64 total processors.

2. By grouping processors correctly, one can apply different processor types to different stages.

3. The inter-processor communication is broken up between the stages, so that

communication between the Preprocessor and Doppler stage can occur while STAP processing is ongoing.

4. The addition of process distribution to the Data-Reorg Library allows the RT_STAP benchmark code to run in both a Clique or Pipelined operation. The benchmark passes validation in both modes.

5. Because the process re-distribution is determined from command line arguments.. This allows use of unmodified code (no recompile) to exercise evaluations. Additionally, the Overloaded MPI library is no longer needed!

6. All the modified code has been sent to MITRE for consideration of inclusion into the future Data Reorg specification.

Refer to Figure 21 for the event log state plot of the new Pipeline mode. The default allocation scheme maps every third processor to the same pool, evenly dividing the work. The processor utilization of this default mapping is not as good as the original Clique mapping, but it is not meant to be. The execution time for 16 iterations is 8.64 seconds with a utilization of 42%. We will see better utilization in the next section after the EGA has computed a better mapping.

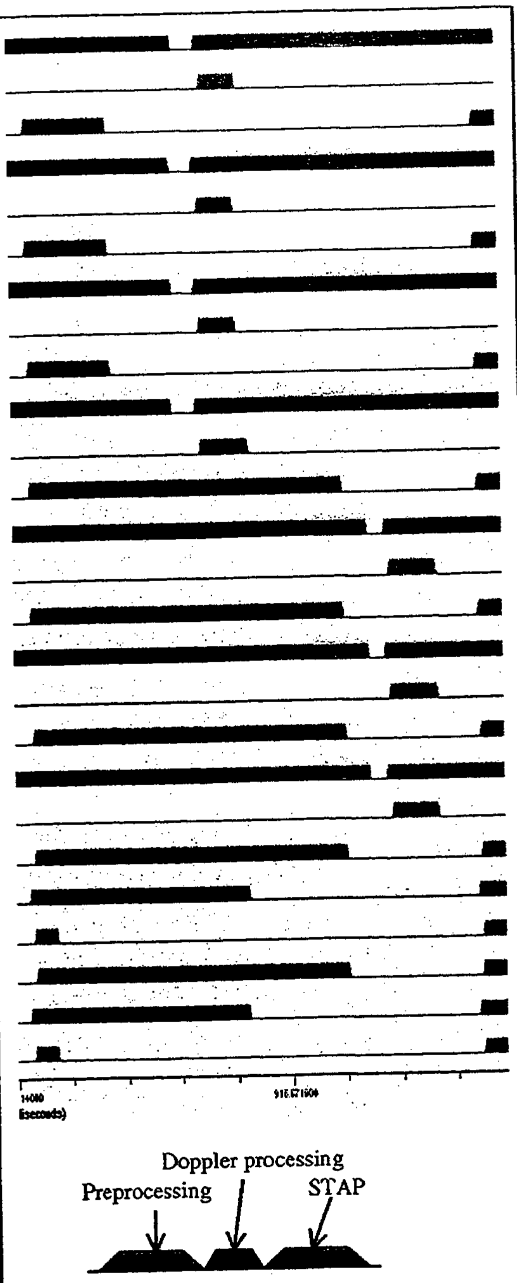


Figure 21-Event Log of RT_STAP in new Pipeline Mode. Default human mapping results in poor processor utilization (worse than Clique).

Modifications to EGA for Optimizing the Data-Reorg Layer

The core EGA remained unchanged for this experiment. However, two major changes occurred:

1) The overloaded MPI library was eliminated. Because processor mapping could be specified through command line parameters, the library was replaced with a series of scripts generated to interface the EGA with the run-time system. These scripts allowed for the automatic generation of the MPI execution script from a given processor allocation. For simplicity, the processor allocation was limited to "pipelined" allocations, wherein the processor groups are completely disjoint.

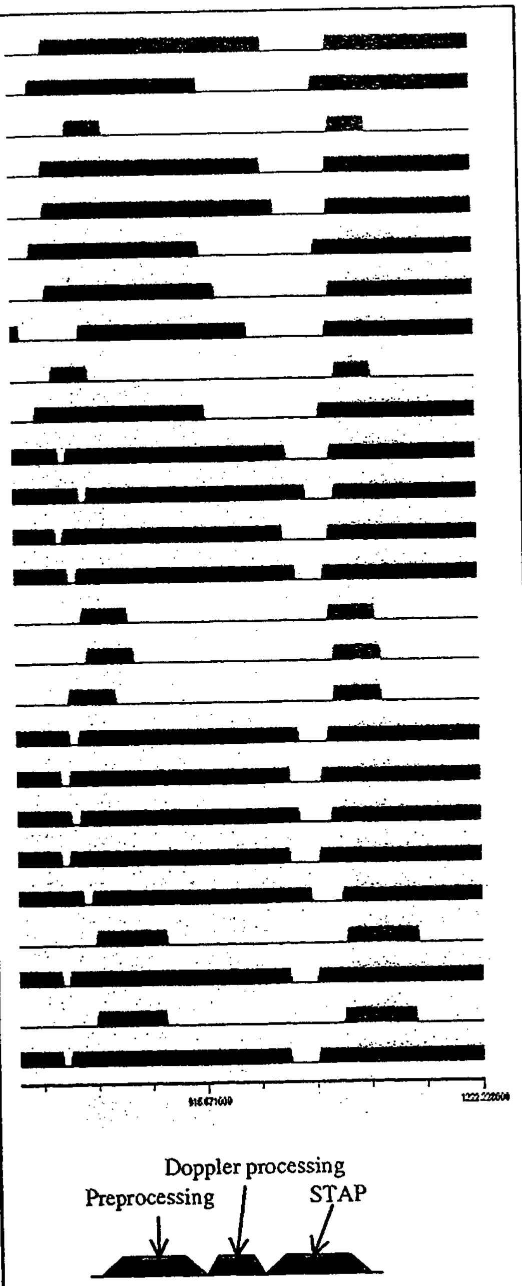


Figure 22-Event Log of RT_STAP Pipeline after EGA Optimization. New mapping gives improved utilization and execution speed (better than Clique).

2) Because larger scale systems were to be used (eventually 128 nodes), EGA was modified to use either an ordered chromosome or an integer chromosome. It was thought that for a larger number of nodes, the Integer might perform better due to the fact that there were fewer possible combinations of processors to explore. A comparison of the optimization time was performed for 63 nodes, as will be shown in a later section.

Demonstration of RT_STAP Optimization – Experimental Results

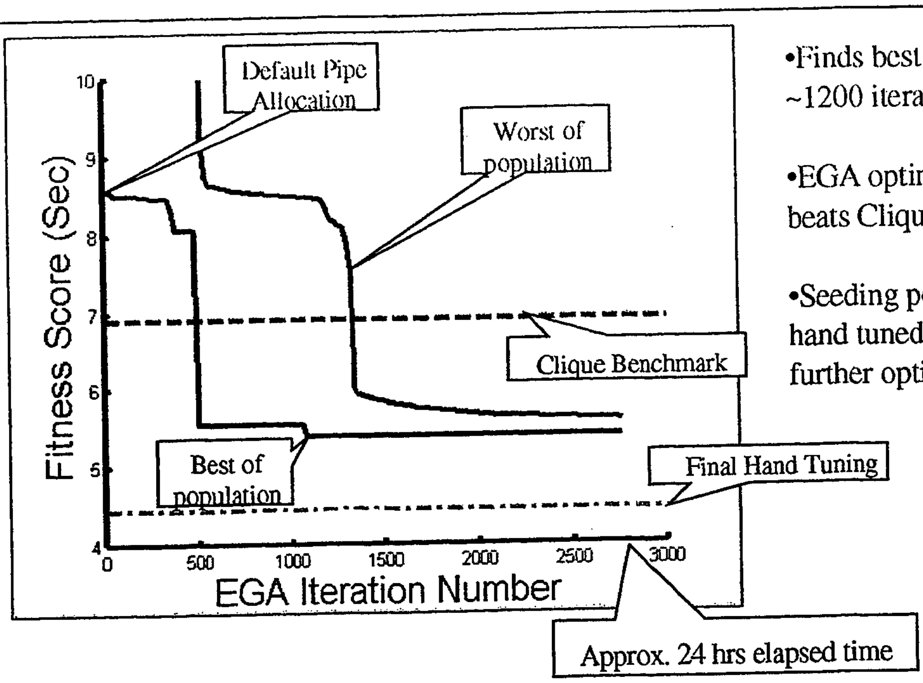
Three experiments were performed using the RT_STAP benchmark. Each one was a larger scale than the next. The first was the RT_STAP MEDIUM[1] problem on a heterogeneous (26-node) system described earlier. The second was the HARD problem on a heterogeneous 63-node system. The last was the HARD problem on a homogenous 128 node Multiport system.

The MEDIUM Problem on a Heterogeneous Mercury RaceWay Multiprocessor

The results of the MEDIUM problem executed on 26 heterogeneous nodes is now presented. The system configuration was described earlier.

The mapping generated by EGA provides 40% processing time improvement over the initial assignment by moving the STAP to PPC processors (where they execute faster) and assigning Pre-Processing to 16 processors (which is a "magic" number for the RT_STAP MEDIUM problem). The 16 Iteration took 5.42 seconds to execute and the resulting utilization is 55%. This mapping beats the Clique

[1] RT_STAP has three data sets: EASY, MEDIUM, and HARD which correspond roughly to the computational load of the problem. The HARD problem does not run on only 28 nodes. The HARD problem is also the one that the community is most interested in.



•Finds best solution after ~1200 iterations (9 hrs)

•EGA optimized Pipeline beats Clique by **20%**

•Seeding population with hand tuned gene resulted in further optimization of 3 %.

Figure 23 - EGA RT_STAP MEDIUM Problem Heterogeneous Experiment - Convergence Properties

implementation by 20% in execution time

The convergence performance of the EGA is as follows. Evaluation of ~ 2750 mapping configurations took about 24 hours. The convergence plot is shown in Figure 23 along with some key details of the run. The EGA converged to a solution (shown as "best of population") after ~1200 iterations (or 9 hrs).

Interestingly enough, when the engineer performing this experiment examined the "best" mapping, he quickly developed a better mapping, taking advantage of the two characteristics that sped up the benchmark. His mapping was far more "ordered" than EGA best was. This resulted in an even faster tuning which performed 16 iterations in 4.44 seconds and had a utilization of 70% (shown in the Figure). When this hand tuned gene was fed back into the population of the EGA, the system found a further optimized mapping which sped up the first corner turn (6% faster) at the expense of a 3% loss of processor efficiency due to using a sub-optimal processor mapping. This gave a total gain of 3% by trading off interconnection bandwidth utilization for processing.

The HARD Problem

After performing the MEDIUM problem optimization, we next tackled the task of optimizing the HARD problem. The hard problem posed new challenges:

1. The data reorganization between the three processing stages required overlapping segments of the data cube to be transferred, thus increasing the interprocessor communication overhead.

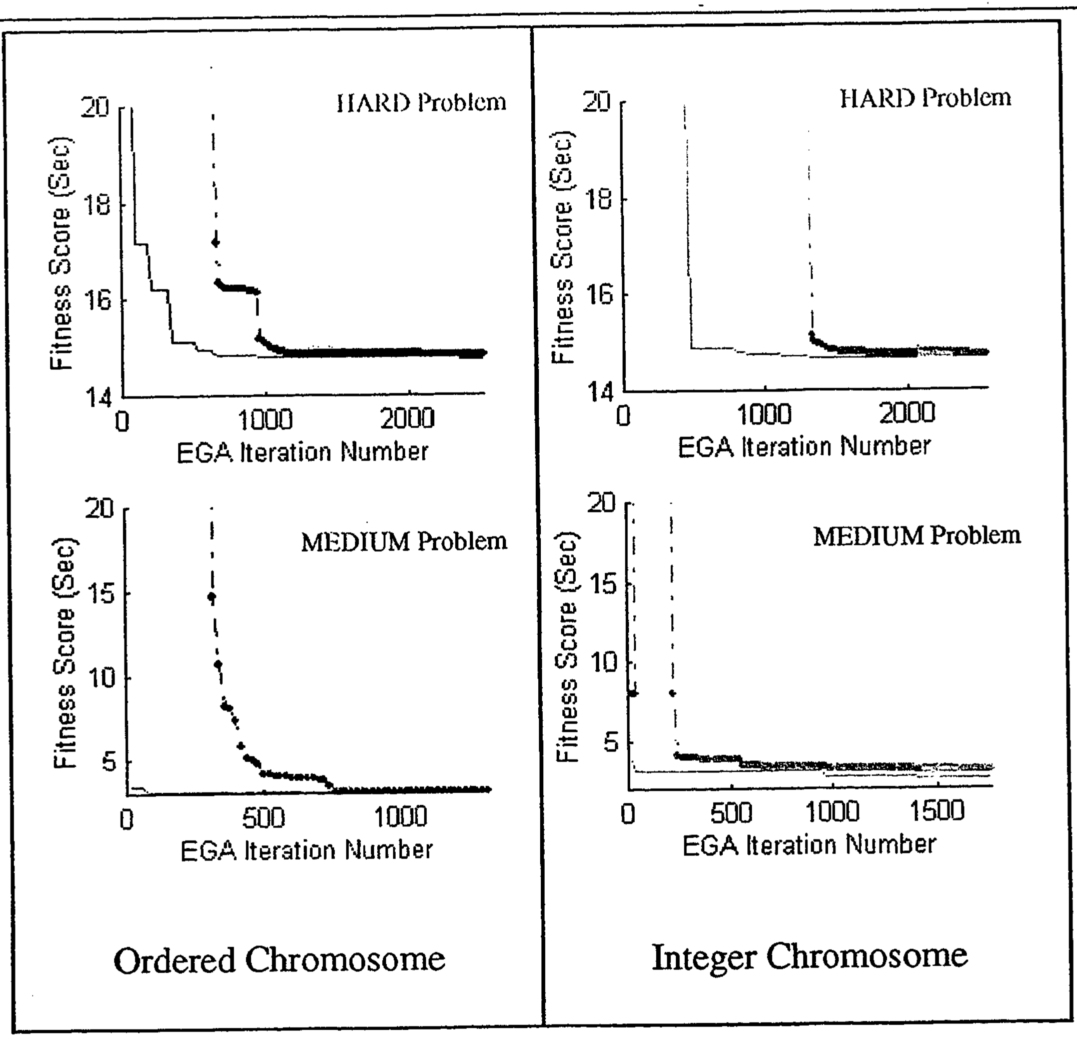


Figure 24 EGA Optimization a 63 node Heterogeneous RT_STAP HARD Problem.

2. The size of the processor pools needed to increase considerably in order to have enough physical memory in the pool to meet the HARD benchmark's memory requirements.

Additionally, in order to run this HARD problem with a rapid iteration time (in order to speed EGA convergence) several additional measures were needed. These included developing code to cache the processor images on reload, and to speed up the download of the input files containing the banchmark data (which was about 35 megabytes in size).

63 Heterogeneous Processor RaceWay Configuration

The HARD problem was run on an augmentation of the original 26 node heterogeneous Mercury multiprocessor. Six PPC nodes and 31[2] i860 nodes were added

[2] The 6 PPC nodes were loaned to us by Mercury Computer, and the 31 i860 nodes were loaned by NUWC Newport. One i860 node on a 16-processor board was broken hence the odd number of nodes. Beggars can't be choosers.

bringing the total number of nodes to 63. The MEDIUM problem was also re-evaluated on the same system. The primary result of this effort was the uncovering and fixing of many bottlenecks to rapid restart and iteration of large Mercury system.

Figure 24 shows the results of four representative runs of the MEDIUM and HARD problem using the Ordered and Integer chromosome representation. . The primary result of the experiment is that the ordered chromosome still gives better convergence performance than the integer chromosome, providing better answers in less time. However, both types of chromosome representations will deliver the same results in the long run. The results of this experiment were not thoroughly analyzed due to arrival of 128-node system.

128 Node Homogeneous Processor Multiport Configuration

Based upon the promise of the results above, Mercury Computer allowed us to utilize their homogeneous 128 node (power pc) Multiport system (called Testdrive). Physically, this system has more interconnection bandwidth per node, so we would expect very homogenous performance in mappings. This was found to be the case. The EGA and RT_STAP programs were transitioned to this new system with minor incident. The only major difficulty arose with using the MPI-PRO version of MPI on this machine. The

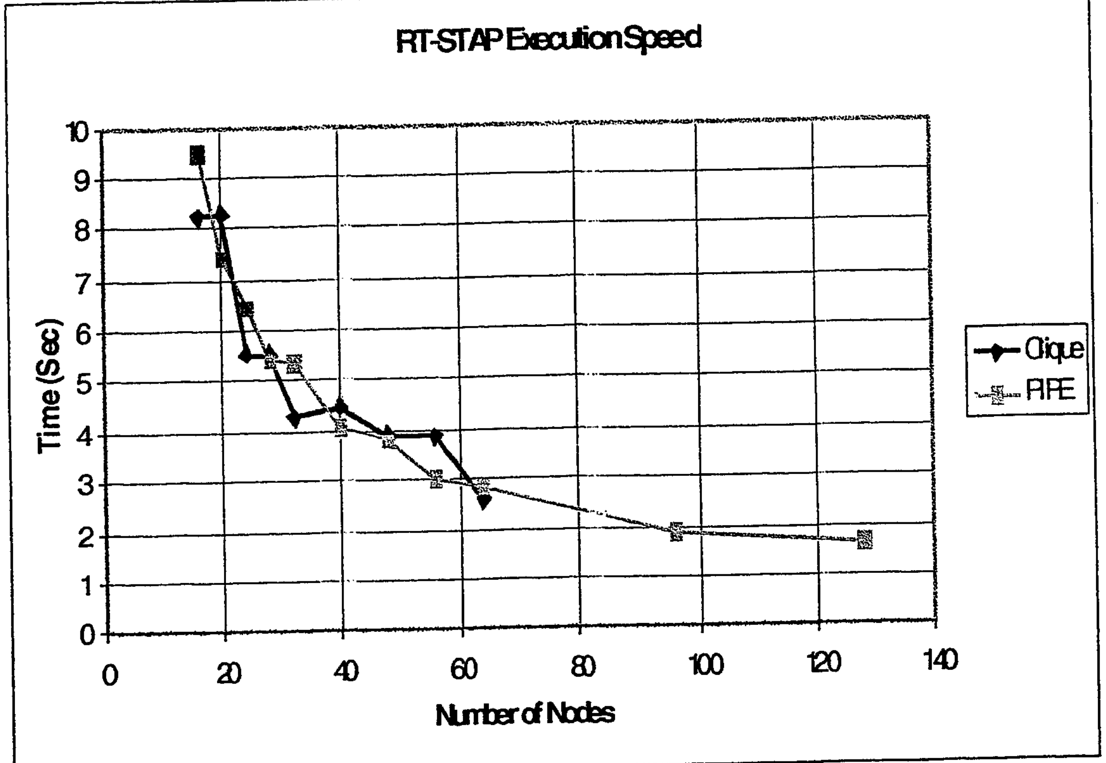


Figure 25 -EGA Optimized Pipeline vs. Clique as a function of total number of processors.

large number of processors, and the repeated iteration of the GA process caused MPI PRO to exhibit many glitches when launched under our EGA scripts. Due to time constraints, we decided to move back to the Hughes version of MPIch, which after minor modification for the newer operating system, performed quite well.

Rather than repeat the same set of experiments once on the larger machine, instead we ran a series of comparisons using different numbers of nodes, in order to see if the EGA optimized pipeline STAP and the original Clique STAP would exhibit any interesting behavior. Figure 25 shows the results of this experiment.

1. Homogeneity of interconnect and processing eliminated the distinct advantage that pipeline RT_STAP had over clique RT_STAP.
2. Pipelining provides finer parallelism, allowing the pipelined benchmark to track a 1/N curve much more closely than the Clique RT_STAP does. However, at magic numbers intrinsic to the code, Clique RT_STAP showed more efficiency, improving in quantized jumps. The only advantage to Pipelining would then be if the desired performance could be achieved by adding a smaller number of nodes (to track the 1/N curve) than would be needed in the Clique case (where more nodes might be needed to reach the jump in performance).
3. The entire mapping process was automated. Thus the system can be retargeted to other sized hosts with no extra design effort.
4. EGA was not stressed by this example (basically it is the performance of the Mercury boot host that limits the iteration time); EGA would be capable of supporting even more flexibility if were provided by the benchmark.

Lessons Learned

The lessons learned in this stage of the project can be summarized as follows:

Overloaded MPI functions are not necessary if the user allows processor mapping to be entered "at the command line". Run-time scripts can then replace the overloaded MPI library for a reduction in system complexity and an increase in portability. The improved Data-Reorg Library hides the resulting program complexity from the user.

For Heterogeneous systems, Pipeline RT_STAP dramatically outperforms Clique RT_STAP, and deals with all the various different node characteristics (processor speed, memory size, I/O) to find global optimum!

For Homogeneous systems, Pipeline STAP performs similarly to Clique STAP, no clear advantage except in two cases"

1. Clique is superior when using "magic numbers" of processors that allow the homogenous MPI task distribution to be done evenly.
2. Pipeline is superior when the "target iteration time" falls between two such "magic numbers", since Pipeline follows a 1/N curve more closely than Clique (which has marked jumps in performance at these numbers).

3. To get even better utilization, future work should move to a function-placement / data-flow system. Though this type of system is even more difficult to code in the conventional manner, using the EGA as an optimizer for the internal mapping layer of a high-level design environment is a very attractive alternative. For example, consider optimizing the internal mapping layer of ISI's RT_ and HRT_express to auto-partition a real time system, and to optimize the placement of code on heterogeneous processors (such as adaptive computing elements or other hardware accelerators.

References

[1] Haban, D. and Wybranietz, D., "A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems," *IEEE Trans. On Software Engineering*, Feb 1990, vol. 16, no 2, pp. 197-211

[2] Roeber, F., "Event Monitoring - The Key to Software Integration," *Proceedings of the Sixth Annual Raytheon Software Symposium*, Jun 1992

[3] Mizar Inc. 2410 Luna Road Carrollton, TX 75006

[4] Davis, L., *Handbook of Genetic Algorithms*, L. Davis Van Nostrand Reinhold, 1991

[5] Holland, J. *Adaptation in Natural and Artificial Systems*, Cambridge, MA: MIT Press, 1992

[6] Mitchell, M., *An Introduction to Genetic Algorithms*, Cambridge, MA: MIT Press, 1996

[7] Tang, K.S., Man, K.F., Kwong, S., and He, Q., "Genetic Algorithms and Their Applications," *IEEE Signal Processing Magazine*, Nov 1996

[8] Cousins, D.; J. Loomis; F. Roeber; P. Schoeppner; and A. Tobin. . "The Embedded Genetic Allocator – A System to Automatically Optimize the Use of Memory Resources on High Performance Scalable, Computing Systems" In *Proceedings of the 1998 IEEE International Conference on Systems, Man and Cybernetics* (San Diego, California, Oct. 11-14 1999). IEEE, Piscataway, N.J., pp. 2166-2171.

[9] Cousins, D.; Daily, M.; Lirakis, C.; Roeber, F. "The Embedded Genetic Allocator – A System to Automatically Optimize MPI Communicator Mappings on High Performance Computing Systems," *In Proceedings of the High Performance Computing Symposium – HPC '99*, at the *Advanced Simulation Technologies Conference* (San Diego, California, April 11-15 1999) Published by the Society for Computer Simulation International, San Diego CA pp. 47-52.

Acknowledgements

This work was supported by the DARPA ITO Embedded Systems Program contract F30602-97-C-0296. We would like to thank Dr. José Muñoz and Col. Mark Swinson of DARPA ITO and Mr. Ralph Kohler of AFRL for their ideas and suggestions. We also would like to thank Mercury Computer Systems, Inc. for their assistance during this project and Rod Ayers of the Naval Underwater Warfare Center, Division Newport for providing the initial algorithms for the Sample Sonar Processor used for the EGA MPI experiments.

RALPH L. KOHLER AFRL/IFTC 26 ELECTRONIC PKWY ROME NY 13441-4514	7
BBN TECHNOLOGIES 10 MOULTON STREET CAMBRIDGE MA 02238	2
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
ATTN: SMDC IM PL US ARMY SPACE & MISSILE DEF CMD P.O. BOX 1500 HUNTSVILLE AL 35807-3801	1
TECHNICAL LIBRARY D0274(PL-TS) SPAWARSYSCEN 53560 HULL ST. SAN DIEGO CA 92152-5001	1
COMMANDER, CODE 4TL000D TECHNICAL LIBRARY, NAWC-WD 1 ADMINISTRATION CIRCLE CHINA LAKE CA 93555-6100	1
CDR, US ARMY AVIATION & MISSILE CMD REDSTONE SCIENTIFIC INFORMATION CTR ATTN: AMSAM-RD-OB-R, (DOCUMENTS) REDSTONE ARSENAL AL 35898-5000	2

REPORT LIBRARY MS P364 LOS ALAMOS NATIONAL LABORATORY LOS ALAMOS NM 87545	1
AFIWC/MSY 102 HALL BLVD, STE 315 SAN ANTONIO TX 78243-7016	1
USAF/AIR FORCE RESEARCH LABORATORY AFRL/VSOSA(LIBRARY-BLDG 1103) 5 WRIGHT DRIVE HANSCOM AFB MA 01731-3004	1
ATTN: EILEEN LADUKE/D460 MITRE CORPORATION 202 BURLINGTON RD BEDFORD MA 01730	1
OUSD(P)/DTSA/DUTD ATTN: PATRICK G. SULLIVAN, JR. 400 ARMY NAVY DRIVE SUITE 300 ARLINGTON VA 22202	1